Optimizing Spatial Min/Max Aggregations

Donghui Zhang
CCIS, Northeastern University
donghui@ccs.neu.edu

Vassilis J. Tsotras*
CSE Dept., UC Riverside
tsotras@cs.ucr.edu

Abstract

Aggregate computation over a collection of spatial objects appears in many real-life applications. Aggregates are computed on values (weights) associated with spatial objects, for example, the temperature or rainfall over the area covered by the object. In this paper we concentrate on MIN/MAX aggregations: "given a query rectangle, find the minimum/maximum weight among all objects intersecting the query rectangle". Traditionally such queries have been performed as range searches. Assuming that objects are indexed by a spatial access method (SAM), the MIN/MAX is computed while retrieving those objects intersecting the query interval. This requires effort proportional to the number of objects satisfying the query, which may be large. A better approach is to maintain aggregate information among the index nodes of the spatial access method; then various index paths can be eliminated during the range search. Yet another approach is to build a specialized index that maintains the aggregate incrementally. We propose four novel optimizations for improving the performance of MIN/MAX queries when an index structure (traditional or specialized) is present. Moreover, we introduce the MR-tree, an R-tree like dynamic specialized index that incorporates all four optimizations. Our experiments show that the MR-tree offers drastic performance improvement over previous solutions. As a by-product of this work we present an optimized version of the MSB-tree, an index that has been proposed for the MIN/MAX computation over 1-dimensional interval objects.

Keywords: spatial aggregates, indexing, Min/Max.

* This research was supported by NSF grants IIS-9907477, EIA-9983445, IIS-0070135, and the Dept. of Defense.

1 Introduction

Computing aggregates over objects with non-zero extent has received a lot of attention recently [YW01, ZMT+01, PKZT01, ZTG02]. Formally, the general *box-aggregation* problem is defined as: "*given n weighted rectangular objects and a query rectangle (box) r in the d-dimensional space, find the aggregate weight over all objects that intersect r*". In this paper we examine the problem of computing the MIN and MAX aggregates (*box-max*) over spatial objects. Each object is represented by its Minimum Bounding Rectangle (MBR) and is associated with a weight (value) that we want to aggregate. Since computing the MIN is symmetric, in the following discussion we focus on MAX aggregation. Moreover, we assume that a spatial access method (SAM) is present (like the R-tree or its variants [Gut84, BKSS90, SRF87].

The box-max problem has many real-life applications. For example, consider a database that keeps track of rainfall over 2-dimensional geographic areas. An example of a box-max query is: "*find the max precipitation in the Los Angeles district*". The database may also keep track of the time intervals of each rainfall, in which case we store 3-dimensional rectangles (one dimension representing the rainfall duration). An interesting box-max query is then: "*find the max precipitation in the Los Angeles district during the interval [1998-2000]*".

Three approaches have been used towards solving box-aggregation queries. The straightforward approach is to simply perform a range search on the SAM that indexes the objects, and compute the aggregation as objects are retrieved. While readily available, this solution requires effort proportional to the number of objects that intersect the query rectangle, which can be large. To improve performance the second approach maintains additional aggregate information on the index nodes of the SAM [JL98, LM01, PKZT01]. For example, the *Aggregation R-tree (aR-tree)* [PKZT01] is an R-tree that stores the aggregate value of each sub-tree in the index record pointing to this sub-tree. Computing an aggregate is now faster since the aggregation information is used to eliminate various search paths.

The third approach utilizes a *specialized* aggregate index built explicitly for computing the aggregate in question [YW01, ZMT+01, ZTG02]. Such an index incrementally maintains the aggregate as objects are inserted/deleted. While this method has the overhead of maintaining an additional structure, the index is usually rather compact (since it does not index the actual data but in practice a much smaller representative set) and provides the best query performance.

The main contributions of this paper are:

- We propose four optimizations for improving the MIN/ MAX aggregation. One of our optimizations (the *k-max*) attempts to eliminate more paths from the index traversal when the aggregate is computed. As such, it can be used either on the SAM that indexes the objects, or, on a specialized aggregate index. The other optimizations (*union*, *box-elimination* and *area-reduction*) eliminate or resize object MBRs when their respective objects do not affect the MIN/MAX computation. Hence they apply only to specialized MIN/MAX aggregate indices.

- We introduce a specialized aggregate index, the *Min/Max R-tree* (MR-tree) that uses all four optimizations. We further present an experimental comparison among a plain R-tree (the R*-tree [BKSS90]), the aR-tree [PKZT01], the aR-tree with the k-max optimization and the MR-tree. Our experiments show substantial improvements when the proposed optimizations are used.

- As a by-product of this research, we discuss how a specialized aggregate index, the MSB-tree [YW00], can be improved by applying the *box-elimination* optimization. The MSB-tree efficiently solves the MIN/MAX problem for the special case of one-dimensional interval objects. The optimization allows the MSB-tree to avoid frequent reconstructions that were needed in its original version.

This work is a full version of [ZT01]. The following material has been added: (i) More detailed and in-depth discussion on the four optimizations. Specifically, section 3.3 which discusses the *union optimization*, contains a new algorithm *CoveredUnion* with analysis; (ii) the complete query and update algorithms of the proposed MR-tree index are presented (section 4.1); (iii) a discussion has been added with insightful understanding of issues affecting the performance of the proposed optimizations (section 4.2); (iv) the experimental evaluation is extended to include a real rainfall dataset; (v) new experiments were introduced examining the choice of index parameters and quantifying the improvement introduced by each individual optimization; (vi) finally, an extended discussion on optimizing the temporal aggregation index (MSB-tree) has been added.

The rest of the paper is organized as follows. Section 2 discusses related previous work. Section 3 identifies the special characteristics of the box-max problem and presents the optimization techniques. Section 4 summarizes the MR-tree while section 5 presents the results from our experimental comparisons. Section 6 discusses the MSB-tree improvement and section 7 presents conclusions.

2 Related Work

There are two variations of the box-aggregation problem, depending on whether objects have zero extent (*point objects*) or not. Aggregation over point objects is a special case of the orthogonal range searching which has received much attention in the field of computational geometry. For more details, we refer to the surveys [Meh84, PS85, Mat94, AE98]. Most of the solutions utilize some variation of the *range-tree* [Ben80] following the multi-dimensional divide-and-conquer technique. Recently, [JL98] proposed the $R_a^* - tree$ which stores aggregated results in the index. [Aok99] proposed to selectively traverse a multi-dimensional index to estimate the selectivity of a range query (which corresponds to the COUNT aggregate). [LM01] proposed the *Multi-Resolution Aggregate Tree (MRA-tree)* which augments the index records of an R-tree with aggregate information for all the points in the record's sub-tree. The MRA-tree also uses selective traversal to provide an estimate aggregation result. The result can be progressively refined. [JL99] proposes a performance model to estimate the performance of index structures with and without aggregated data.

A special case of the point aggregation problem is the work on data cube aggregation for OLAP applications. A data cube [GBLP96] can be thought of as a multi-dimensional array. [RKR97] proposed the *cubetree* as a storage abstraction of the cube and realized it using packed R-trees to efficiently support cube and group-by aggregations. [HAMS97] addressed both the box-max and the *box-sum* (for SUM, COUNT and AVG) queries over data cubes. The solution to the box-max query was based on storing precomputed max values in a balanced hierarchical structure. This solution was further improved by [HAMT97]. The solution to the box-sum query was based on pre-computing the *prefix sum*, which is the aggregate over a range covering the smallest cell of the array. This solution was improved by [GAAS99, CI99, GAA00]. Recently, [CCLL01] proposed the *dynamic update cube* which further improves the update cost to $O(\log k_u)$ where k_u is the number of changed array cells.

For aggregations over objects with non-zero extents, [YW01] presented the *SB-tree* which solves the box-sum query in the special case of one-dimensional time intervals. The SB-tree was extended to the *Multi-version SB-tree (MVSB-tree)* in [ZMT$^+$01] to efficiently support temporal box-sum aggregation queries with key-range predicates. [ZTG02] addressed box-sum aggregation over spatiotemporal objects. Furthermore, [YW00] presented the *MSB-tree* for the box-max query over one-dimensional interval objects. The aR-tree [PKZT01] was originally proposed to index the spatial dimension in a spatial data warehouse environment, but can be used to solve both the box-sum and the box-max queries over spatial data with non-zero extent. Similarly with the MRA-tree, the aR-tree is an R-tree which stores for each index record the aggregate value for all objects in its sub-tree. Since the aR-tree was built for the support of both box-sum and box-max queries, it is not fully optimized towards box-max queries. The aR-tree is used here as a starting point for our optimizations and it is included in our experimentation for comparison purposes. Also related is [AS90] which answers window queries on top of the *pyramid data structure*. Aggregations are used for the existence/non-existence of image features and the visibility in terrain data. Last, in the spatial-temporal data warehouse environment, [PTKZ02] proposed the *aggregate R-B-tree (aRB-tree)* which uses an R-tree to index the spatial dimension while each data record in the R-tree has a pointer to a B-tree which keeps historical data about this record.

3 The Proposed Optimizations

We first formally define the **box-max** problem which we address in this paper.

> *Maintain a set of spatial objects, each of which has a rectangular region (box) and a value, so that for any query box q, find the max value among all objects that intersect q.*

Assume that an R-tree is used to index the spatial objects. A straightforward approach to solve the box-max problem, is to perform a range-search for all objects whose extent intersects the query region, and pick the max value among these objects. Instead, the aR-tree proposed in [PKZT01] stores along with every index entry, the max value of all objects in the sub-tree. If the query region completely contains the MBR of a sub-tree, this sub-tree needs not be examined. By incorporating four optimization techniques into the aR-tree approach, we can achieve substantial (orders of magnitude) query performance improvement.

We first introduce some notations. An index/leaf record is an entry in an internal/leaf node of the aR-tree. Given a leaf record r, let $r.box$ and $r.value$ denote the MBR and the value of the record, respectively. Given an index record r, let $r.box$ denote its MBR, $r.value$ denote the maximum value of all records in subtree(r) and $r.child$ denote the child page pointed by r.

3.1 The k-max optimization

The aR-tree can only omit examining a sub-tree if it is *fully* contained in the query box. However, index records at higher levels of the tree, have typically large MBRs. As a result, the box-max query is not likely to stop at higher levels of the aR-tree.

We hereby propose the k-max optimization which makes it possible omitting a sub-tree even if the query box does not contain the MBR of the sub-tree. The idea is shown in Figure 1. Here the biggest box corresponds to the MBR of some sub-tree while the shadowed box illustrates the query. Since the query box does not contain the MBR of the sub-tree, the aR-tree approach cannot prune the sub-tree. However, if along with the index entry which points to the sub-tree, some max-valued objects are stored, the sub-tree can be pruned. For instance, suppose we store the three objects shown in the numbered boxes (the numbers are the object values). These objects should be max-valued in the sense that for any remaining object in the sub-tree, the value is no more than 9 (the smallest of the stored values). It can be derived from this example, that the sub-tree will contribute a value 18 to the query, without actually browsing the sub-tree.

The k-max optimization *Let k be a small constant. Along with each index record r, store the k objects which are in subtree(r) and have the largest values among the objects in the subtree. When examining record r during a box-max query, if the query box intersects with any of the k max-value objects in r, the examination of subtree(r) is omitted.*

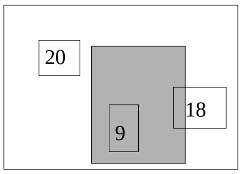


Figure 1: **Illustration of the k-max optimization.**

Clearly, the k-max optimization allows for more paths to be omitted during the index traversal. Nevertheless, the benefit of k-max on the query performance has an overhead. The overall space is increased (since each node stores more information) as well as the update time (extra effort is needed to maintain the k objects). Hence in practice the constant k should be kept small. In our experiments, substantial improvement in query time was observed even for $k = 3$.

The remaining optimizations apply to an index explicitly maintained for the MIN/MAX aggregation (to avoid confusion we call such a specialized index the MIN/MAX index). Since the MIN/MAX problem is not incrementally maintainable when tuples are deleted from the database [YW01] the following discussion assumes an append-only database (i.e., spatial objects are inserted in the database but never deleted). When a spatial object o with MBR $o.box$ and value $o.value$ is inserted in the database, $o.box$ accompanied by $o.value$ is inserted as a leaf record in the MIN/MAX index as well. However, as we will describe, some of these insertions may not be applied to the MIN/MAX index, or even may cause existing MBRs to be deleted or altered from the MIN/MAX index. As such, we use an R*-tree [BKSS90] to implement the MIN/MAX index. The result after applying all four optimizations will be the MR-tree.

3.2 The box-elimination optimization

Consider two leaf records o_1 and o_2, where $o_1.box$ contains $o_2.box$ and $o_1.value \geq o_2.value$. There is no need to maintain o_2 in the MIN/MAX index since it will not contribute to any MAX query. We thus say that o_2 becomes *obsolete* due to o_1, or o_1 makes o_2 obsolete. For example, in Figure 2 object o_2 can be removed without affecting the query correctness. The reason is that any query (the shadowed region) which intersect o_2 will also intersect o_1, and thus the query result is at least as large as $o_1.value$.

In general, an object o_1 can eliminate a whole sub-tree, if $o_1.box$ contains the MBR of the sub-tree and $o_1.value$ is no less than the max value stored in the sub-tree.

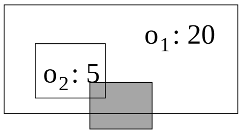


Figure 2: **Illustration of the box-elimination optimization.**

The box-elimination optimization *If during the insertion of an object o, a (leaf or index) record r is found such that o.box contains r.box and o.value* $\geq$ *r.value, remove r from the MIN/MAX index; if r is an index record, remove subtree(r) as well.*

The above optimization will reduce the size of the MIN/ MAX index, since sub-trees may be removed during an insertion. There is a tradeoff between the time to update the MIN/MAX index and the overall space occupied by this index. A newly inserted object may make obsolete more than one existing records which are on different paths from the root to leaves. The MIN/MAX index can be made very compact if all these obsolete records (and their sub-trees) are removed. However, this may result in expensive update processing. To maintain fast updates, we can remove only the obsolete records met along the insertion path (which is a single path since the MIN/MAX index was implemented using the R*-tree). The complexity of the insertion algorithm remains $O(\log(n))$ where n is the number of MBRs in the MIN/MAX index (which in practice is much smaller than the total number of spatial objects in the collection). Another choice is to choose c paths and remove the obsolete records met along these paths, where c is a constant. The space occupied by the obsolete sub-trees can be re-used.

3.3 The union optimization

While the box-elimination optimization focuses on making obsolete existing records in the MIN/MAX index, the *union* optimization focuses on making obsolete objects before they are inserted in the tree. Note that the MBR of an object should not be inserted in the MIN/MAX index if there is an existing leaf object in the index whose MBR contains it and has a larger value. Such an insertion can be safely ignored for the purposes of MIN/MAX computation. To fully implement this test, all the paths that may contain this object have to be checked; at worst this may check all leaf objects in the MIN/MAX tree. A better heuristic is to use the k max-value MBRs. If the new object is contained by any of the k max-value MBRs found along the index nodes in the insertion path, and has a smaller value, then the insertion can be safely omitted.

Moreover, even if the MBR of an object to be inserted is not fully contained by any existing leaf object, it may still be safely ignored. This is the case when the new object's MBR is contained in the union of MBRs of several existing objects. As illustrated in Figure 3, the shadowed box represents the new object to be inserted and the other two rectangles represent two objects already in the MIN/MAX index. Since the new object is contained in the union of the two existing objects with a smaller value, this object is obsolete and does not need to be inserted.

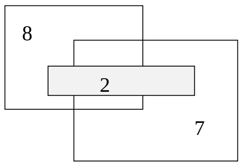


Figure 3: **The new object becomes obsolete by the union of two existing objects.**

To implement this technique, each index record r in the MIN/MAX index stores (1) the union (denoted by $r.union$) of MBRs of all the leaf objects in subtree(r), and (2) the minimum value (denoted by $r.low$) of all the objects in the subtree(r). The overall optimization is described below:

The union optimization *If during the insertion of object o, an index(leaf) record r is found such that o.box is covered by r.union(r.box) and o.value $\leq$ r.low(r.value), the insertion is ignored. Moreover, the insertion of o is obsolete if there exists some max-value object stored in r that covers o.box and has a value no smaller than o.value.*

A remaining question regarding the above optimization is how to store the union of all leaf objects under an index record. At worst, this union may need space proportional to the number of leaf objects that create it. Given that each index record has limited space, we store an *approximate* union. In particular, we store a good approximation that can be represented with t boxes (MBRs), where t is a small constant. Clearly, the approximate union should be completely covered by the actual union, so that the approximation affects only the query time, but not the correctness of the query result. If the approximate union covers some area not covered by the actual union, it may erroneously make some new object obsolete. This approximate union is called the *covered t-union*, which we formally define as follows.

Definition 1 *Given constant t and a set of n boxes $S = \{s_1, \cdots, s_n\}$ where $n \gg t$, the* **covered t-union** *of S is defined as a set of t boxes $\{a_1, \cdots, a_t\}$ such that: (1) $\cup_{i=1}^{n} s_i$ covers $\cup_{i=1}^{t} a_i$; and (2) $\cup_{i=1}^{t} a_i$ is maximal, i.e. there does not exist another set of t boxes $\{a'_1, \cdots, a'_t\}$ covered by $\cup_{i=1}^{n} s_i$ such that $\cup_{i=1}^{t} a'_i$ covers larger space than $\cup_{i=1}^{t} a_i$.*

An exhaustive search algorithm for finding the covered t-union works as follows: (1) determine an exhaustive collection of sets, where each set has t boxes, while making sure that the covered t-union belongs to the collection; (2) exclude every set from the collection such that the union of the set's t boxes are not covered by the original n boxes; and (3) among all sets in the collection, choose the one such that the union of its t boxes covers the largest area.

However, it can be shown that the above algorithm takes $O(n^{2t+4})$ time, which makes it impractical. Below we propose an efficient $O(n \log n)$ algorithm which computes a approximation of the covered t-union.

Algorithm *CoveredUnion*(Boxes *source*[1..n]) Given a set of n boxes *source*, return an approximate covered t-union of *source*. Below t, c and *max_try* are small constants.

```
1.  Let seeds be the c·t boxes from source whose areas are the largest;
2.  Initialize the set of destination boxes dest to be empty;
3.  for i from 1 to t
4.      Pick the box b from seeds which has the largest area not covered by the union of boxes in dest;
5.      loop max_try times
6.          Try to enlarge b along every dimension;
7.      endloop
8.      Add b to set dest;
9.  endfor
10. return dest;
```

The idea behind algorithm *CoveredUnion* is to pick t boxes from the original n boxes and try to expand each one of them as much as possible. The i^{th} box ($1 \leq i \leq t$) is chosen as the one having the largest area not covered by the $i-1$ boxes computed so far. To enlarge a chosen box b along some dimension, say the X dimension (step 6 of the algorithm), the higher x-value of b is increased and the lower x-value of b is decreased, while ensuring that the enlarged box is still covered by the original n boxes. Clearly, any corner p of a box in the covered t-union should satisfy the following condition: the x-value of p is equal to the x-value of a corner of some box in *source*, and the y-value of p is equal to the y-value of a corner of some box in *source*. Thus, the higher x-value of box b is increased to the nearest larger x of some corner of one of the *source* boxes. To implement this, the algorithm should have a preprocessing step which sorts the $2n$ x-values of corners in the *source* boxes and similarly for the y-values. The complexity of the algorithm is $O(n \log n)$.

To better understand the CoveredUnion algorithm, we compare it with the following straightforward algorithm (the *top-t algorithm*), that picks from the n source boxes, the t boxes whose areas are the largest. The rationale of the top-t algorithm is as follows: (a) it is easier to pick boxes from the source boxes than to construct some new boxes; and (b) since the goal here is to maximize the area covered by the t boxes, a heuristic is to pick the boxes whose areas are the largest. It can be seen that the top-t algorithm also returns t boxes covered by the union of the source n boxes. Nevertheless, we illustrate below that the CoveredUnion algorithm can find a better solution in the sense that the area covered by its t boxes is at least as large as the area covered by the boxes identified from the top-t algorithm.

To see this, consider an even simplified version of the CoveredUnion algorithm, where boxes are not enlarged (i.e. by removing steps 5-7). This simplified CoveredUnion produces better results than the top-t algorithm. Let's order the t boxes reported by the top-t algorithm in decreasing order of new area covered. That is, the first chosen box has the largest area; the second box is chosen from the rest t-1 boxes such that, if it is combined with the boxes already chosen, the total covered area is larger than (or equal to) the case when any other box is chosen; and so on. The simplified CoveredUnion algorithm also picks t boxes in order and adds them to the *dest* set. The first box chosen is the same as that picked by the top-t algorithm. However, for any subsequent box chosen, the increase in new area covered is at least as large as the area increase brought from the box picked by the top-t algorithm. This statement can be verified by step 4 of the CoveredUnion algorithm.

In the following discussion the term *union* means the approximate covered t-union computed by the CoveredUnion algorithm.

3.4 The area-reduction optimization

The last optimization we propose dynamically reduces the box area of the object to be inserted.

The area-reduction optimization *If during the insertion of object o, an index record r is found such that $r.union$ intersects with $o.box$ and $r.low \geq o.value$, we reduce the size of $o.box$ by subtracting the area covered by $r.union$ from it before inserting it to the lower index levels. If the insertion reaches a leaf object which intersects the new object and has an equal or larger value, the area of the new object is reduced accordingly. The box of the new object is similarly reduced if some max-value object stored in an index record r exists whose box intersects with $o.box$ and whose value is no smaller than $o.value$.*

This optimization reduces the MBR of an object only if the reduced part is covered by some existing records in the tree with larger or equal values. Hence the correctness of the MIN/MAX aggregates is not affected. One benefit of this optimization is that overlapping among sibling records in the tree is reduced. Figure 4 shows an example. The two large boxes represent two index records r_1 and r_2. Assume $r_1.union$ is equal to the MBR of r_1. The combination of the light-shadowed and the dark-shadowed boxes represents an object to be inserted with value 8. The object should be recursively inserted into subtree(r_2). Without applying the area-reduction optimization, $r_2.box$ would need to be expanded to fully contain the new object. On the other hand, if the optimization is applied, the light-shadowed area is subtracted and thus a much smaller box (the dark-shadowed area) is inserted in subtree(r_2). This smaller box is fully contained in $r_2.box$ and thus no expansion for r_2 is needed.

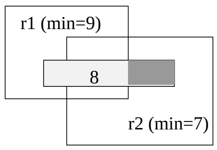


Figure 4: **The area-reduction optimization reduces overlaps.**

Another benefit of the area-reduction optimization is that it helps to make new records obsolete. As an example, consider Figure 4 again. It is possible that at some lower level in subtree(r_2), the dark-shadowed area is found obsolete. Since the light-shadowed area is already made obsolete by r_1 due to the optimization, there is no need to insert the record (with value 8) at all.

Note that the result of a box when some areas are subtracted from it may be a set of boxes rather than a single box. Hence this optimization may fragment an inserted object into several smaller boxes. One approach would be to insert every small box as a separate object. But this choice increases the space overhead. Another choice is to maintain the list of small boxes in the execution of the insertion algorithm. As the tree is traversed, some small boxes may become smaller or obsolete. Eventually at the leaf level, the MBR of the remaining smaller boxes is inserted. Note that the MBR is at most as large as the original box to be inserted and in many cases, much smaller.

4 The Min/Max R*-tree

The MR-tree is a dynamic, disk-based, height-balanced tree structure. There are two types of pages: leaf pages and index pages. All pages have the same size. Since the MR-tree is based on the R*-tree, each page except the root has at most M records and at least m records. Each leaf record has the form $\langle box,\ v_1\rangle$, where v_1 is the value of the record. Each index record has the form $\langle box,\ child;\ b_1,\ v_1, \cdots,\ b_k,\ v_k;\ union,\ low\rangle$, where *box* and *child* have their usual meanings. The list $(b_1,\ v_1), \cdots,\ (b_k,\ v_k)$ is the k max-value leaf objects in the sub-tree of this index record, sorted by decreasing order of value. Here b_i stands for the MBR and v_i for the value of the leaf object with the i^{th} largest value. The *union* stores t boxes (the approximate t-union over all leaf MBRs), and *low* is the minimum value over all leaf objects in the sub-tree.

4.1 The Query and Insertion Algorithms

We proceed with the range query and insertion procedures for the MR-tree.

Algorithm *BoxMax*(Page N, Box b, Value v) Given a tree node N, a query box b and a running value v, the algorithm returns the box-max query result for the sub-tree rooted by N.

1. for every record r in N where $r.box$ intersects with b
2.   if $r.v_1 > v$ then
3.   if N is leaf then
4. $v = r.v_1$;
5.   else if there exists i in $[1,\ k]$ such that $r.b_i$ intersects with b
6.   Let i be the smallest one satisfying this condition;

```
7.              if r.v_i > v, set v = r.v_i;
8.          else
9.              v =BoxMax(Page(r.child), b, v);
10.         endif
11.     endif
12. endfor
13. return v;
```

To compute the box-max over box b, procedure BoxMax(root page, b, $-\infty$) is called. $BoxMax$ is a straightforward recursive algorithm. The main difference between this algorithm and the range query algorithm in an R-tree is in steps 5-7, which correspond to the k-max optimization. For an index record r, the algorithm checks the k max-value objects stored in r. If any of them intersects with b, there is no need to examine subtree(r).

Algorithm *Insert*(Tree T, Box b, Value v) Given tree index T, a box b and a value v, the algorithm inserts an object with b and v into T.

```
1.  S = {b};
2.  N =root page of T;
3.  while ( N is not leaf ) do
4.      for every record r in N where r.box intersects with b
5.          if r.box is contained in b and r.v_1 ≤ v then
6.              Remove subtree(r);
7.          else
8.              for every i such that r.v_i ≥ v
9.                  Modify each box in S by subtracting r.b_i from it;
10.             endfor
11.             if r.low ≥ v, modify every box in S by subtracting r.union from it;
12.         endif
13.     endfor
14.     if N has zero record, goto step 19;
15.     if S is empty, goto step 20;
16.     N =ChooseChild(N, MBR(S));
17. endwhile
18. Optimizations for a leaf page; similar to steps 4 through 13; omit;
19. if S is not empty, insert ⟨MBR(S), v⟩ into N;
20. while ( N is not root ) do
21.     if N overflows then
22.         Split(N);
23.     else if N underflows then
24.         Remove N and reinsert the records from N into the tree at N's level;
25.     endif
26.     Adjust the entry in Parent(N) pointing to N;
27.     Set N =Parent(N);
28. endwhile
```

```
29. if N overflows then
30.     Split old root and create a new root;
31. else if N has only one record and N is not leaf then
32.     Remove N and set N's child as the new root;
33. endif
```

Generally, the insertion algorithm follows a single path from the root down to a leaf. Reorganizations may follow the path back up to the root. The optimizations are applied when traversing down the tree. Steps 5 and 6 correspond to the box-elimination optimization which removes a sub-tree if the newly inserted object has a larger value and spatially contains the sub-tree. Steps 8 to 11 correspond to the area-reduction optimization which tries to reduce the size of the box to be inserted. Step 14 deals with the rare case when all sub-trees in some index page N become obsolete due to the insertion of an object. This may occur only when N is a root page, since otherwise the index record pointing to N in the parent page would be obsolete before N has a chance to be examined. For this case, the algorithm results in a tree with a single page and a single record. Step 15 means that the object to be inserted is obsolete and no recursive insertion into lower levels are needed. Step 16 chooses a child page to recursively insert into. The same algorithm as in the R*-tree is used, thus the *ChooseChild* procedure is not discussed in detail. In steps 20 to 28, the insertion path is examined backwards, from leaf up to root. Splitting an overflowed page and entry reinsertion in an underflowed page are identical to the approaches in the R*-tree, plus the maintenance of the additional information kept along with each index record. Steps 29 through 33 handle overflow/underflow of the root page. As a consequence, the tree height may increase/decrease.

4.2 Discussion

We proceed with a discussion comparing the worst-case behavior of the MR-tree with the aR-tree. The measures examined are (1) index size, (2) query performance, and (3) index creation time.

First, consider the k-max optimization. If this optimization is applied alone, the index size and index creation time will increase due to storing extra information along with each index entry (k max-valued objects). However, since k is a constant, the fan-out of the MR-tree is asymptotically the same as that of the corresponding aR-tree. That is, the index size is asymptotically the same as for the aR-tree, and so is the index creation time. On the other hand, it is possible that the query performance is improved vastly. At best, the query box may intersect with one of the k max-valued objects kept in the root of the MR-tree. In this case, only the root node of the MR-tree needs to be accessed for answering the query. Even in the worst case, i.e. when no pre-computed object reduces query processing, the MR-tree query performance is asymptotically the same with the aR-tree approach.

When considering the box-elimination optimization, there are two choices. If the objects/sub-trees are eliminated only along the insertion path, the optimization can improve all three measures. In the worst case, no elimination occurs. However when there is elimination, e.g. of a sub-tree, the index size shrinks, which in return improves both query performance and index creation time (smaller size index). The second choice is to further compact the tree by checking multiple paths to find opportunities to eliminate sub-trees. Compared with the first choice, the index size and query performance remain equal or improve, but the index creation time increases. Since the number of paths examined is a constant, this increase is bounded. That is, the worst-case index creation time for the second choice is asymptotically the same as for the first.

However, the index size improvement of the second choice may not be significant to justify in practice the extra index update time. The rationale is as follows. The R*-tree tries to cluster objects based on spatial proximity. Hence if all the paths from root to leaf in an MR-tree are considered, where it is possible to eliminate some sub-tree covered by a new object, chances are that the insertion path is the best. Equivalently, most existing entries that can be removed from the existing tree are likely to be located within the insertion path. For this reason, we advocate using the first choice of only examining the insertion path. This approach does not need extra I/O to perform the box-elimination checking since the object must be insert into the tree anyways.

Similar to the k-max optimization, the union optimization maintains some extra information along with each index entry in the MR-tree (the covered t-union). Maintaining the extra information may increase in practice (although asymptotically remains the same) the index size and index creation time. However, this increase will not be large since at the same time a smaller number of objects is inserted decreasing the index size.

Finally, the area-reduction optimization tends to shrink the area of new objects which in turn reduces the overlap between sibling nodes in the tree, and increases the possibility to eliminate the insertion of an object. Hence this optimization improves all the three measures.

In summary, we make following observations:

- The worst-case performance (index size, query, index creation time) of the MR-tree is asymptotically the same as the aR-tree.
- For datasets where it is unlikely that an object spatially contains another object, out of the four optimizations only the k-max optimization will be helpful. This will improve query performance (against the aR-tree), but we will increase index size and index creation time.
- For datasets with frequent object containments (i.e., large object overlapping), all four optimizations should be applied. This will lead to a much smaller index, since many objects are either not inserted, or inserted but later removed. Even though additional information is kept along with each index entry, it is likely that the MR-tree will be is superior in all measures (index size, query time and index creation time).

In the following section we evaluate these observations through experimentation.

5 Performance

We compare the performance of the proposed MR-tree against the plain R*-tree, the aR-tree and the aR-tree with the k-max optimization (denoted as *aR-tree*$_{kmax}$). In the following figures we use *MR, R*, aR* and *kaR* respectively, to represent the various methods.

5.1 Experimental Setup

We experimented with two synthetic datasets (sections 5.2, 5.3) and a real dataset (sections 5.4, 5.5). Their characteristics are summarized in Table 1. In particular, for the two synthetic datasets, the space is assumed to have size 1 million by 1 million units. Within this area, 5 million square objects were generated randomly. For the *High-Overlap Set*, the edge size of each object was randomly chosen from 10 to 10,000. For the *Medium-Overlap Set*, the edge size of each object

Datasets	**Description**	**Overlap**
High-Overlap Set	Synthetic dataset of 5 million objects, edge size in [10, 10,000]	86.1%
Medium-Overlap Set	Synthetic dataset of 5 million objects, edge size in [10, 1000]	29.3%
Low-Overlap Set	Real dataset of 20,499 objects	15%

Table 1: Dataset Description.

was randomly chosen from 10 to 1000. In the High-Overlap dataset it is more likely that objects spatially contain other objects since the object sizes tend to vary substantially. Analysis showed that in this dataset 86.1% of the objects were contained by other objects. In the Medium-Overlap dataset, objects tend to have similar sizes and thus the amount of overlap is reduced. Here 29.3% of the objects were contained by other objects.

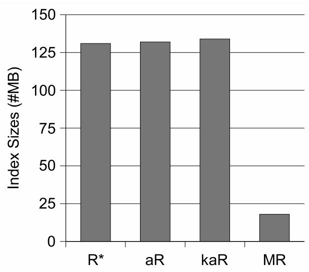
We also experimented with a real dataset denoted as *Low-Overlap Set*. It was derived from the rainfall dataset [Iri03] maintained by the International Research Institute for Climate Prediction (IRI). IRI provides monthly rainfall precipitation of the whole Earth from January 1979 to September 2003. IRI acquired data by using rain gauges (there are over 8000 gauges all over the world), satellite, and numerical model predictions. The Earth is divided into 144*72=10368 grid cells, where each cell corresponds to a 2.5 degree longitude by 2.5 degree latitude region. For every month, a precipitation value is provided per region. From the original data, we generate rectangular objects using the following transformation. First, for each cell and month, if the precipitation is less than 0.003m, we treat it as zero. Second, for each month, we examine the 2-d array of precipitation values and identify rectangular regions with non-zero precipitation, and we take the average value of cells in each rectangle as the value of the object. There were 20,499 rectangular objects generated. Thus generated objects tend to be large and have similar sizes. Only 15% of the objects are contained by other objects.

All algorithms were implemented in C++ using GNU compilers. The experiments were run on a Dell Optiplex GX260 PC with a 2.66-GHz Pentium 4 processor. The OS is CYGWIN_NT-5.1 running on top of WinXP Professional. By default, for the MR-tree we chose $k = t = 3$, and for the aR-tree$_{kmax}$ we chose $k = 3$.

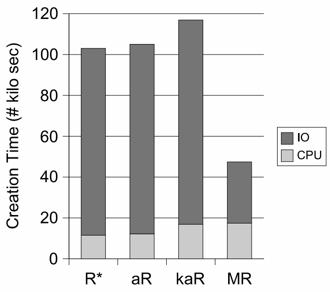
Each index utilizes an LRU buffer and a *path buffer*, which buffers the most recently accessed path. For the two synthetic datasets, the total memory buffer we used for each program has 256 pages, and each disk page size is 4KB. For the real dataset, since the number of objects is relatively small (20.5K), we chose 1KB page size, and 100-page buffer size.

5.2 Experiments with the High-Overlap Dataset

Figure 5 compares the index sizes and the index creation time for the High-Overlap Dataset. The MR-tree has much smaller index sizes than the other index structures, since most objects can be eliminated from the index. To measure the index creation time, we count each page fault as 10ms and we added the computed I/O time to the measured CPU time. Here the CPU time was measured by adding the time spent in *user* and *system* mode as returned by the *getrusage* system call. The MR-tree has also the best index creation time. We note that the improvement on index creation time of the MR-tree is not as significant as the improvement on index size. This is because the MR-tree spends extra computation time corresponding to the optimizations.



(a) Index sizes



(b) Index creation time

Figure 5: **Index size and index creation time comparison for the High-Overlap Dataset.**

For the query performance evaluation, the query rectangle area varies from 0.0001% to 50% of the whole space. For each query rectangle size, 100 square queries (of the same size) were randomly generated and the *total* running time was measured in number of I/Os. Clearly, the MR-tree has the best performance. Note that the I/O scale is logarithmic in this figure. The reason for the improved performance is that for large query rectangles, the MIN/MAX query has more chances to stop at higher levels in the MR-tree. In particular, the aR-tree will decide to omit examining a sub-tree only if the query rectangle contains the box of the whole sub-tree. On the other hand, the MR-tree search may omit traversing a sub-tree even if the query rectangle partly intersects with it. The usefulness of the k-max optimization can be seen when comparing the aR-tree$_{kmax}$ with the plain aR-tree. The MR-tree performs better than the aR-tree$_{kmax}$ for two reasons. First, due to the additional optimizations, the MR-tree stores fewer objects. Second, objects in the MR-tree have smaller area (the area reduction optimization), and thus better clustering is achieved.

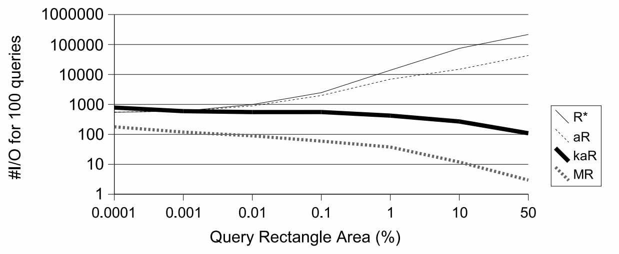
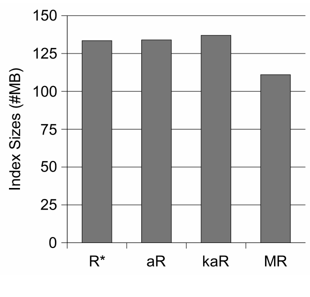


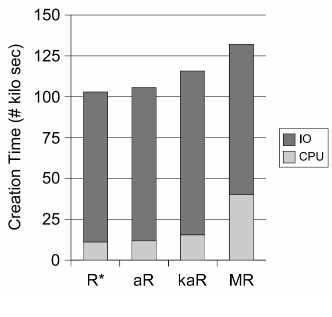
Figure 6: **Query performance for the High-Overlap Dataset.**

5.3 Experiments with the Medium-Overlap Dataset

Figure 7 compares the index size and index creation time for the Medium-Overlap dataset. The MR-tree used about 20% less space (Figure 7a) than the other methods. This is because some obsolete records were removed from the index, but the percentage of obsolete objects is smaller than in the High-Overlap dataset. The aR-tree$_{kmax}$ occupies the most space, since when compared with the R*-tree and the aR-tree, it stores more information in each index record. The MR-tree has about 25% more index creation time than its competitors (Figure 7b) since more objects take part in the optimizations.



(a) Index sizes



(b) Index creation time

Figure 7: **Index size and index creation time comparison for the Medium-Overlap Dataset.**

The query performance is shown in Figure 8. While all methods are comparable for small query rectangles (since few objects satisfy the query anyways), the MR-tree is clearly the best as the query rectangle size increases.

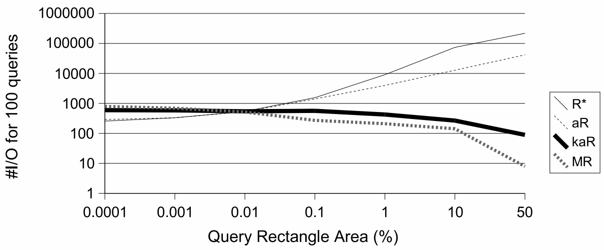
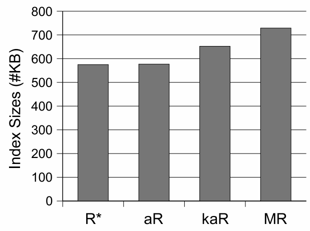
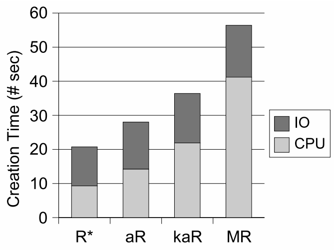


Figure 8: **Query performance for the Medium-Overlap Dataset.**

5.4 Experiments with the Low-Overlap Dataset



(a) Index sizes



(b) Index creation time

Figure 9: **Index size and index creation time comparison for the Low-Overlap Dataset.**

As Figure 9 shows, for the Low-Overlap dataset the MR-tree index size is larger than the other methods (by about 22%). All methods show substantial increase in index creation time against the R*-tree. The MR-tree showed the largest increase since for this dataset, its optimizations did not omit many objects.

The query performance comparison is shown in Figure 10. Similar to the experiments with synthetic datasets, we observed that the query performance of the MR-tree is better as the size of the query rectangle increases. However, since in this dataset not many objects can be omitted, the query rectangle has to be fairly large in order for the optimizations to become effective.

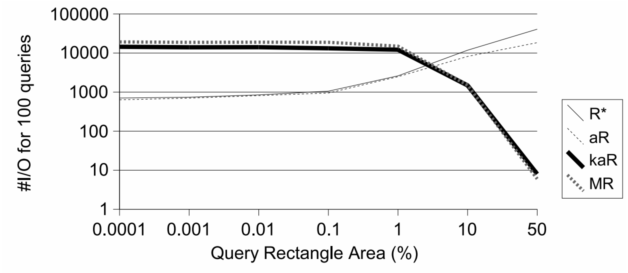
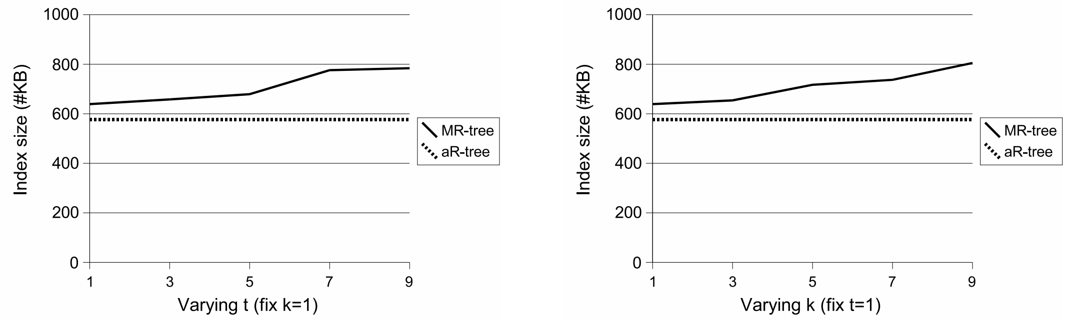


Figure 10: **Query performance for the Low-Overlap Dataset.**

5.5 Optimizing the Tree Parameters

In this section we further optimize the performance of the MR-tree by varying its k and t parameters from the k-max and the covered t-union optimizations respectively. We use the performance of the aR-tree as a baseline comparison. Given that the Low-Overlap dataset provided the less improvement for the MR-tree (i.e., a "worst-case" environment), we use this dataset for the optimization experiments. The goal is to understand how different choices of the above parameters affect the index performance. For example, the values of both k and t affect the index size, since more records are stored in the index. Given some space constraint it is interesting to see how these parameters should be chosen, that is, whether a larger k or a larger t would provide better query performance?



(a) (b)

Figure 11: **Comparing the MR-tree index size while varying t and k.**

Figure 11 compares the effect of varying k and t on the MR-tree index size. In Figure 11(a), five indices are created for the MR-tree, with fixed k (k=1) while varying t from 1 to 9. Similarly, in Figure 11(b), t was fixed (t=1) and k was varied. As expected, in both cases the size of the

MR-tree index increases as we increase either k or t. This is due to the storage of extra $k + t$ objects/rectangles for each index entry in the MR-tree. It seems that for this rainfall dataset, not a lot of objects were omitted.

Next we examine the effect in query performance. In Figure 12, various values of t are depicted while $k = 1$, and for different query rectangle sizes. For each query size, 100 random queries with the same size were generated and the total execution time was measured in number of disk I/Os (the CPU time was much smaller compared with the I/O time). Similarly, Figure 13 compares the query performance for different query sizes while varying k ($t = 1$).

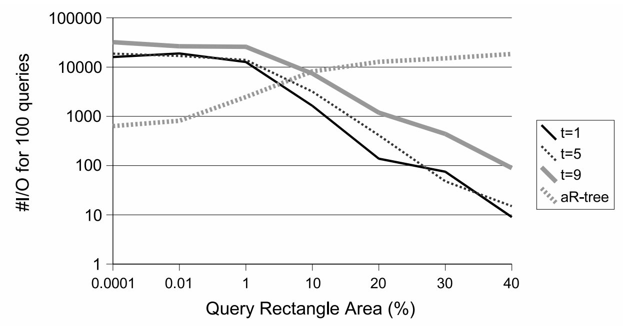


Figure 12: **Query performance for varying query rectangle sizes (the MR-tree has $k = 1$ while t varies.**

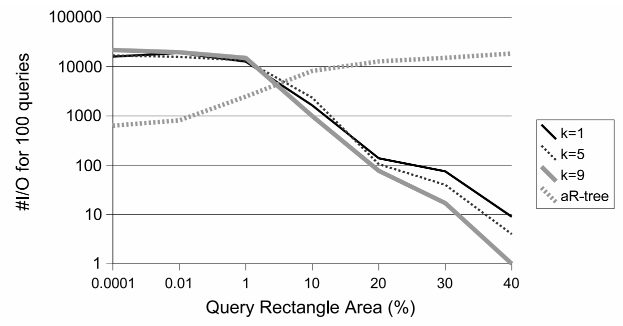


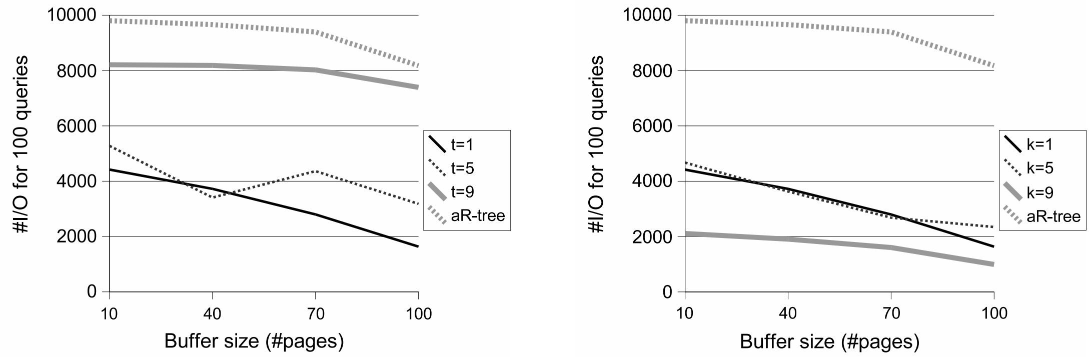
Figure 13: **Query performance for varying query rectangle sizes (the MR-tree has $t = 1$ while k varies.**

From Figures 12 and 13 we make the following observations:

(a) The MR-tree's performance improves as the query area increases. The reason is that when the query size is small, the aR-tree search space is small. So not too many leaf pages need to be searched. On the other hand, since the MR-tree has a smaller fan-out (each index entry is augmented with some additional information), more index pages need to be accessed. But for a large query rectangle, the MR-tree has better performance as the search may very likely stop at higher levels.

(b) The query performance improves while increasing k and decreasing t. Higher k increases the possibility that the query rectangle intersects with one of the k objects maintained at a high level index entry. This in turn improves query performance. To explain while increasing t does not improve performance note that the t boxes were used as an approximate covered t-union to omit the insertion of new objects. A new object can be omitted only if the minimum value in the sub-tree is larger than the value of the new object. And chances are this may not be the case as t increases.

We also compared the query performance while varying the buffer size. Here the query size is chosen to be 10% of space. Again, we observe that increasing k is more preferable than increasing t.



(a) Varying t (b) Varying k

Figure 14: **Query performance with different buffer sizes.**

A summary of our experimental findings follows:

- As expected, for datasets where objects experience large overlap, the MR-tree tends to have smaller index size as more objects may be omitted.
- The query time of the MR-tree decreases as the query size increases (opposite to the R-tree or the aR-tree).

- Increasing k is more preferable than increasing t.
- We recommend using small k and t, since larger k and t means large size for an index entry, which reduces the fan-out of the tree. In practice we obtained good performance with values of k between 1 and 10, and t between 1 and 3.

6 Optimizing the MSB-tree

We first review the MSB-tree which was proposed by [YW00] to answer the box-max queries for 1-dimensional interval data. We then show how one of our proposed optimization techniques (the box-elimination optimization) can be utilized to improve it.

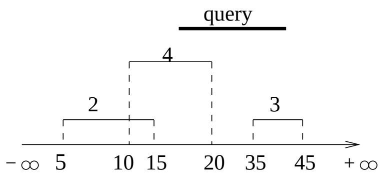


Figure 15: **Illustration for three objects and a query in the 1-dimensional case.**

In order to support box-max for the 1-dimensional data, [YW00] proposed to pre-compute the aggregates by dividing the time space into segments and associating a value with each segment. For example, if there are three objects as shown in Figure 15, the time space can be divided into six segments as follows:

[-∞,5]:-∞, [5,10]:2, [10,20]:4, [20,35]:-∞, [35,45]:3, [45,+∞]:-∞.

Here each segment is represented by [start,end]:value, where value=-∞ means that not part of the corresponding interval has been covered by an inserted object yet.

An example box-max query as illustrated by the thick line segment in Figure 15 has result 4, since the query interval intersects two objects and the larger value of the objects is 4. Equivalently, the query can be performed by scanning through the above six segments and return the largest value among the intervals which intersect the query.

When there are too many such segments that they do not fit in one disk page, a B+-tree like index (the MSB-tree) is maintained: adjacent segments are clustered into leaf pages of the index.

Along with the index record pointing to a leaf page, an interval and a value are maintained. The interval is the union of all segments in the page, and the value is the max value across all segments in the page. If there are a lot of such index records, they are clustered into index pages recursively. At the highest level, the MSB-tree has a single root.

To perform a query on the MSB-tree, the root page is examined first. We know that the intervals (of records) in the page are connected to each other. So among the intervals that intersect the query, there can be at most two whose intervals partially intersect the query, while all the others are fully contained in the query. If the root page is an index page, for any record in the page whose interval is fully contained in the query, there is no need to examine the sub-tree referenced by this record. The reason is that the max value of the segments in the sub-tree is already stored along with the index record. For each of the two records whose intervals partially intersect the query, the query algorithm recursively goes to the page references by the record. Note that this time, each page can have at most one interval which is partially contained in the query. Thus the query algorithm follows two paths from the root level to the leaf level and thus the complexity of the algorithm is $O(\log_B m)$, where B is the page capacity and m is the number of leaf-level elements.

The index as described above may have expensive updates, since each update (inserting an object with an interval and a value) need to examine all the leaf elements which intersect the inserted interval. To improve performance, [YW00] proposed to maintain an additional value along with each index record as the min value of the leaf-level elements in the sub-tree. By doing so, we achieve $O(\log_B m)$ update performance as well. The reason is that for any index record r whose interval is contained in the interval of the object o to be inserted, there is no need to update the sub-tree referenced by this index record. Instead, we update the min value stored at r to be the larger one between $o.value$ and the original min value of r.

An MSB-tree is called *compact* if the number of leaf records in it is minimum. The MSB-tree update algorithm does not ensure the compactness of the tree. Since it is ideal to maintain a small m, [YW00] proposed to periodically reconstruct the MSB-tree. To reconstruct, the whole tree is browsed in a depth-first manner to report every interval together with the aggregation value during this interval. The intervals thus reported are continuous to one another. Adjacent intervals with the same value are merged. All the intervals are then inserted into a second, initially empty MSB-tree which eventually replaces the original tree. During the reconstruction phase, the MSB-tree is *off-line*, i.e. no new insertion can be made.

We discuss how to apply the box-elimination optimization (proposed in section 3) to the MSB-tree to get a relatively much smaller tree while it remains on-line. The idea is that whenever the min and max values of an index record r becomes equal during an insertion process, remove the whole sub-tree referenced by r. The min and max value of r will become equal when an new object is inserted whose interval contains the interval of r and whose value is larger than the value of r. Note that in a MSB-tree, the union of intervals of all records at each level is the whole space. However, if a sub-tree is removed from the index, this condition does not hold any more. We need to adjust the tree such that the condition still holds. Suppose an index record r points to the sub-tree which was removed. If r is the only record in a page, merge the page with a sibling page; otherwise, there exists a sibling record s where the sub-tree pointed by s has not been removed yet (otherwise r and s would have been merged into one record). Without lose of generality, suppose s is a right sibling of r. Merge r with s by extending the interval of s to contain $r.i$. Also, the first record of every node in the leftmost path starting from $s.child$ needs to be extended as well.

To analyze the complexity of the modified algorithm, note that at each level of the tree, at most two pages are examined. In each page, there are $O(B)$ obsolete records. For each of these records, we need to follow a single path from the page containing the record to some leaf. Thus the worst case update complexity is $O(B \log_B m)$ where m is the number of leaf records. This discussion does not count the cost to free up the space occupied by the sub-trees pointed to by obsolete records. In fact, since these sub-trees are not needed in any update or query to be performed later, the garbage collection can be performed by a background process. Compared with the original MSB-tree update algorithm, the modified update algorithm is a little more expensive in the worst case. However, this does not happen often, since each time the modified algorithm spends more time in update, the tree is shrunk.

The major benefit of the optimized algorithm is that it results in a much smaller tree without periodic reconstruction. Consider the insertion of an object o whose interval is the whole space and whose value is larger than that of every existing record. The original algorithm simply updates the min and max values of all root-level records and thus the number of leaf records does not change.

On the other hand, the optimized algorithm immediately decides that the whole tree is obsolete and thus results in a very compact tree: a tree with only one leaf record.

7 Conclusions

We examined the problem of computing MIN/MAX aggregation queries over spatial objects with non-zero extents. We proposed four optimization techniques for improving the query performance. We introduced the MR-tree, a new index explicitly designed for the maintenance of MIN/MAX aggregates. The MR-tree combines all proposed optimizations. Experimental comparison showed that our approach provides drastic improvement especially when query sizes increase. As a by-product, we showed how one of the optimizations can be applied to improve an existing aggregation index (the MSB-tree).

Acknowledgments

We would like to thank D. Gunopulos for many helpful discussions, D. Papadias for providing valuable input on related work and B. Seeger for the R*-tree code. We also thank the anonymous referees whose comments have improved the paper.

References

[AE98] P. Agarwal and J. Erickson. Geometric Range Searching and Its Relatives. In B. Chazelle, E. Goodman, and R. Pollack, editors, *Advances in Discrete and Computational Geometry*. American Mathematical Society, Providence, 1998.

[Aok99] P. M. Aoki. How to Avoid Building DataBlades that Know the Value of Everything and the Cost of Nothing. In *Proceedings of International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 122–133, 1999.

[AS90] W. G. Aref and H. Samet. Efficient Processing of Window Queries in the Pyramid Data Structure. In *ACM International Symposium on Principles of Database Systems (PODS)*, pages 265–272, 1990.

[Ben80] J. L. Bentley. Multidimensional Divide-and-Conquer. *Communications of the ACM*, 23(4):214–229, 1980.

[BKSS90] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data (SIGMOD)*, pages 322–331, 1990.

[CCLL01] C. Chung, S. Chun, J. Lee, and S. Lee. Dynamic Update Cube for Range-Sum Queries. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, pages 521–530, 2001.

[CI99] C. Chan and Y. E. Ioannidis. Hierarchical prefix cubes for Range-Sum Queries. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, pages 675–686, 1999.

[GAA00] S. Geffner, D. Agrawal, and A. El Abbadi. The Dynamic Data Cube. In *Proceedings of International Conference on Extending Database Technology (EDBT)*, pages 237–253, 2000.

[GAAS99] S. Geffner, D. Agrawal, A. El Abbadi, and T. Smith. Relative Prefix Sums: An Efficient Approach for Querying Dynamic OLAP Data Cubes. In *Proceedings of International Conference on Data Engineering (ICDE)*, pages 328–335, 1999.

[GBLP96] J. Gray, A. Bosworth, A. Layman, and H. Piramish. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. In *Proceedings of International Conference on Data Engineering (ICDE)*, pages 152–159, 1996.

[Gut84] A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data (SIGMOD)*, pages 47–57, 1984.

[HAMS97] C. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range Queries in OLAP Data Cubes. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data (SIGMOD)*, pages 73–88, 1997.

[HAMT97] C. T. Ho, R. Agrawal, N. Megiddo, and J. J. Tsay. Techniques for Speeding up Range-Max Queries in OLAP Data Cubes. *IBM Research Report*, 1997.

[Iri03] NOAA NCEP CPC Merged Analysis monthly December 2003 Release (Version 1). *International Research Institute for Climate Prediction, URL=http://iridl.ldeo.columbia.edu/SOURCES/.NOAA/.NCEP/.CPC/.Merged_Analysis/.monthly/.v0312/*, 2003.

[JL98] M. Jürgens and H. J. Lenz. The R_a^*-tree: An Improved R-tree with Materialized Data for Supporting Range Queries on OLAP-Data. In *International Workshop on Database & Expert Systems Applications (DEXAW)*, 1998.

[JL99] M. Jürgens and H. J. Lenz. PISA: Performance Models for Index Structures with and without Aggregated Data. In *Proceedings of International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 78–87, 1999.

[LM01] I. Lazaridis and S. Mehrotra. Progressive Approximate Aggregate Queries with a Multi-Resolution Tree Structure. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data (SIGMOD)*, pages 401–412, 2001.

[Mat94] J. Matoušek. Geometric Range Searching. *ACM Computing Surveys*, 26(4):421–461, 1994.

[Meh84] K. Mehlhorn. Multi-dimensional Searching and Computational Geometry. *Data Structures and Algorithms*, 3, 1984.

[PKZT01] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient OLAP Operations in Spatial Data Warehouses. In *Proceedings of Symposium on Spatial and Temporal Databases (SSTD)*, pages 443–459, 2001.

[PS85] F. Preparata and M. Shamos. *Computational Geometry: An Introduction*. Sprinter Verlag, 1985.

[PTKZ02] D. Papadias, Y. Tao, P. Kalnis, and J. Zhang. Indexing Spatio-Temporal Data Warehouses. In *Proceedings of International Conference on Data Engineering (ICDE)*, pages 166–175, 2002.

[RKR97] N. Roussopoulos, Y. Kotidis, and M. Roussopoulos. Cubetree: Organization of and Bulk Incremental Updates on the Data Cube. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data (SIGMOD)*, pages 89–99, 1997.

[SRF87] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-Tree: A Dynamic Index for Multi-dimensional Objects. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, pages 507–518, 1987.

[YW00] J. Yang and J. Widom. Temporal View Self-Maintenance. In *Proceedings of International Conference on Extending Database Technology (EDBT)*, pages 395–412, 2000.

[YW01] J. Yang and J. Widom. Incremental Computation and Maintenance of Temporal Aggregates. In *Proceedings of International Conference on Data Engineering (ICDE)*, pages 51–60, 2001.

[ZMT+01] D. Zhang, A. Markowetz, V. J. Tsotras, D. Gunopulos, and B. Seeger. Efficient Computation of Temporal Aggregates with Range Predicates. In *ACM International Symposium on Principles of Database Systems (PODS)*, pages 237–245, 2001.

[ZT01] D. Zhang and V. J. Tsotras. Improving Min/Max Aggregation over Spatial Objects. In *ACM International Symposium on Advances in Geographic Information Systems (GIS)*, pages 88–93, 2001.

[ZTG02] D. Zhang, V. J. Tsotras, and D. Gunopulos. Efficient Aggregation over Objects with Extent. In *ACM International Symposium on Principles of Database Systems (PODS)*, pages 121–132, 2002.